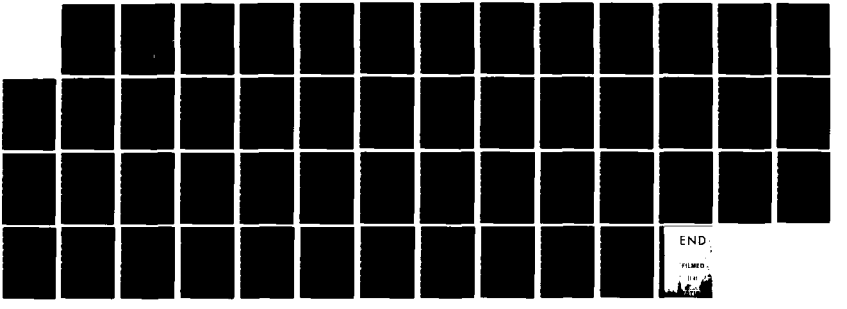
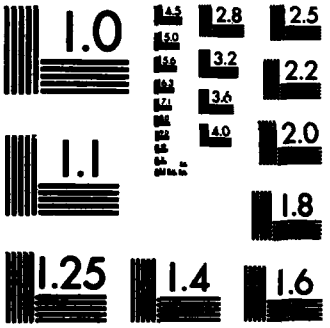
AD-A134 091 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA INTEGRATED ENVIRONMENT.. (U) INTERMETRICS INC CAMBRIDGE MA 08 OCT 82 IR-MA-142-1 F30602-80-C-0291 1/1

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

IR-MA-142-1
COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR Ada INTEGRATED ENVIRONMENT: VIRTUAL MEMORY METHODOLGY B5-AIE (1).VMM (2)

8 OCTOBER 1982

CONTRACT F30602-80-C-0291

PREPARED FOR: ROME AIR DEVELOPMENT CENTER
CONTRACTING DIVISION/PKRD
GRIFFISS AFB, N.Y. 13441

PREPARED BY: INTERMETRICS, INC.
733 CONCORD AVE.
CAMBRIDGE, MA 02138

This document was produced under contract F30602-80-C-0291/SA P0009 for the Rome Air Development Center. Mr. Douglas White is the Program Engineer for the Air Force. Mr. Mike Ryer is the Project Manager for Intermetrics.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

FIGURES

1.0 SCOPE

1.1 Identification

This specification defines the requirements for the Virtual Memory Methodology subsystem (CPCI AIE.VMM.VMM). VMM is a component of the Minimal Ada Programming Support Environment (MAPSE) of the Ada Integrated Environment (AIE), providing MAPSE tools with facilities required to construct and manipulate data in a consistent, reliable, and portable form.

1.2 Functional Summary

MAPSE tools must be able to preserve data structures within the Kernel Ada Programming Support Environment (KAPSE) Database in order to communicate with other tools or with subsequent activations of the same tool. In general, it cannot be assumed that the address space of the host machine will be adequate to keep such data structures entirely within memory while they are used. A Virtual Memory Methodology provides both a means of representing the data structures used by tools in a consistent and efficiently-accessed external form, and a means of overcoming address space limitations on the size of data structures. In addition, VMM provides aids to debugging and communication between hosts (ASCII, human-readable forms).

The VMM package consists of four major components (CPC's):

1. The Rep Analyzer (VMM.VMM.A), which converts user-defined data structures into VMM structures and creates a VMM access package via which user tools can access data;

2. The ABSTRACT DATA TYPES Package (VMM.VMM.B), which defines standard data aggregates such as lists;

3. The VRN IO Package (VMM.VMM.C), which converts input and output to or from a linear, human-readable form;

4. The BASIC OPERATIONS Package (VMM.VMM.D), which implements the basic operations (e.g., create, delete, access) on VMM structures.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

2.0 APPLICABLE DOCUMENTS

2.1 Program Definition Documents

Requirements for Ada Programming Support Environments, "STONEMAN", February 1980, Department of Defense.

Revised Statement of Work, 15 March 1980.

Reference Manual for the Ada Programming Language, proposed standard document, U.S. Department of Defense, 1980.

2.2 Inter Subsystem Specifications

System Specification for Ada Integrated Environment, Type A.

Computer Program Development Specifications for Ada Integrated Environment, Type B5:

Ada Compiler Phases, AIE(1).COMP(1)

KAPSE/Database, AIE(1).KAPSE(1)

MAPSE Command Processor, AIE(1).MCP(1)

Program Integration Facilities, AIE(1).PIF(1)

MAPSE Debugging Facilities, AIE(1).DBUG(1)

MAPSE Text Editor, AIE(1).TXED(1)

Technical Report (Interim)

Computer Program Product Specification for Ada Integrated Environment, Virtual Memory Methodology, Type C-5, AIE(1).VMM(2).VMM.

2.3 Military Specifications and Standards

Data Item Description DI-E-30139, USAF, 14 July 1976.

2.4 Miscellaneous Documents

Diana Reference Manual, G. Goos and Wm. Wulf, Institut Fuer Automatik II, Universitaet Karlsruhe and Computer Science Dept, Carnegie-Mellon University, March 1981.

Guido Persch, Georg Winterstein, Manfred Dausmann, Sophia Drossopoulu, AIDA Implementation Description, Institut fuer Automatik II, Universitaet Karlsruhe, December 1980.

Wetherell, Charles, Alfred Shannon, LR: Automatic Parser Generator and LR(1) Parser, Preprint UCRI-82926, University of California, Davis, June 1979.

Pager, D., A Practical General Method for Constructing LR(k) Parsers, Acta Informatica 7, pp.249-268, 1977.

Manfred Dausmann, Sophia Drossopoulu, Guido Persch, Georg Winterstein, An Informal Introduction to AIDA, Institut fuer Automatik II, Universitaet Karlsruhe, November 1980.

Benjamin M. Brosgol, David Alex Lamb, David R. Levine, Joseph M. Newcomer, Mary S. Van Deusen, William A. Wulf, TCOLAda: Revised Report on an Intermediate Representation for the Preliminary Ada Language, Computer Science Dept, Carnegie-Mellon University, February 1980.

R. Cattell, D. Dill, P. Hilfinger, S. Hobbs, B. Leverett, J. Newcomer (principal editor), A. Reiner, B. Schatz, W. Wulf, PQCC Implementor's Handbook, October 1980.

J. D. Ichbiah, J. G. P. Barnes, J. C. Helian, B. Krieg-Bruechner, O. Roubine, B.A. Wichmann, Rationale for the Design of the Ada Programming Language; ACM SIGPLAN Notices, Vol. 14, No.6, June 1979, Part B.

Intermetrics LG Description, 31 August, 1980, IR-536

LG User's Guide, December 1979, IR-427

YACC - Yet Another Compiler Compiler, Johnson, Stephen, C.7, Documents for the PWP/UNIX Time-Sharing System, Edition 1.0, October 1977.

3.0 REQUIREMENTS

3.1 Introduction

The VMM (Virtual Memory Methodology) subsystem is a tool for creating and manipulating abstract data structures (attributed directed graphs, in particular) in a machine-independent manner. A virtual memory paging scheme makes the size of any data structure independent of the memory constraints of any particular hardware configuration. The Virtual Record Notation IO package (VRN IO) supports input and output of data in an external, human-readable format, translating data into the terms of the particular implementation thereby satisfying portability requirements. A permanent, directly accessible representation of a VMM data structure can be created in the KAPSE database and accessed by any MAPSE tool or user program, providing a standard data interface for inter-program communication.

VMM is written in Ada and incorporated into the AIE as normal Ada code (using the Rep Analyzer). The implementation packages are also written in Ada and compiled into the program library. VMM will be used to implement the CPCI, COMP.DIANA.

3.1.1 General Description

VMM provides support for:

1. Data Structure Definition. VMM translates Ada package specifications defining the data structures required by a particular tool, producing a template which maps the data structures onto KAPSE database objects when the tool is run.

2. Virtual Memory Management. VMM performs automatic paging to provide any tool with a directly addressable memory area well beyond the actual addressing range of the host system.

3. Machine-Independent Data. Automatic conversions between a directly accessible representation of a data structure and either of two host-system-independent linear representations is a basic VMM facility. A user may obtain:

   a) a human-readable text describing the structure's concrete realization, which is primarily of use for low-level debugging and testing; or

   b) a human-readable text describing the structure's abstract properties, similar to the concrete form but reflecting the logical, rather than physical, structure (see 3.3.4).

3.1.2 Peripheral Equipment Identification

As do all MAPSE tools, VMM relies on the KAPSE for all hardware interfacing. See [B-5-AIE(1).KAPSE(1)].

3.1.3 Interface Identification

VMM's primary software interfaces are: the KAPSE, the user (tool builder), and the Ada compiler.

3.2 Functional Description

3.2.1 Equipment Descriptions

Not applicable.

3.2.2 Computer Input/Output Utilization

Not applicable.

3.2.3 Computer Interface Block Diagram

Not applicable.

3.2.4 Program Interfaces

3.2.4.1 KAPSE Interface

VMM interfaces with the KAPSE database via package DIRECT_IO in PREDEFINED PACKAGES in KAPSE.RTS. VMM instantiates the package for a single record type (which is known as a VMM page) and creates dynamic objects that include objects of the type FILE_TYPE from that instantiation.

The string names for database objects, required by certain Input/Output operations, are obtained in one of the following ways:

a) as actual parameters to VMM operations that invoke Input/Output operations;

b) as values from database objects whose names were previously supplied as actual parameters to VMM operations; or

c) by defaulting the filename parameter (for temporary files).

VMM also operates on FILE_TYPE objects of package TEXT_IO in PREDEFINED PACKAGES in KAPSE.RTS. These files are used for error messages and human-readable representations.

3.2.4.2 User Interface

Since VMM is a method for implementing data structures used by MAPSE tools, a "user interface" to the tool builder is a major consideration. This interface has three main components:

a) data structure definitions;

b) primitive operations; and

c) linear representations.

3.2.4.2.1 Data Structure Definitions

Data structure definitions describe the data structures to be accessed by a given tool. These definitions must provide sufficient information so that the tool can access the data as an Ada object, and so that VMM operations can be derived to create the object, store and retrieve it in an "external file", and/or convert it to or from a human-readable representation. To accomplish this, the tool builder uses an Ada package specification to define a virtual record type that describes the structure of the data and the allowable data types for components of the structure. This virtual record type is specified as an Ada variant record type with a single discriminant. The discriminant must be of an enumeration type defined in the same package.

A virtual record is named by a literal belonging to the enumeration type, and consists of those components that are applicable to the variant record having that value as a discriminant. This is a basis for mapping between internal and human-readable forms.

The types of virtual record components are limited to a subset of Ada type constructors and a set of types supplied by the VMM implementation through generic packages. In particular, no Ada ACCESS types may be used; instead, VMM locator types are used.

While the definition package specification coded by the tool builder is valid Ada, it is not directly compiled with the tool. Instead, the specification is processed by a program referred to as the Rep Analyzer; this combines the definitions for one or more virtual record types and enforces the restrictions and conventions required by VMM. The Rep Analyzer generates as output: a new package specification and package body for each virtual record type (called the virtual record type declaration); and a package specification and body called the access package that provides access to one or more virtual record definitions.

Operations on VMM data that are available to the MAPSE tool are declared in the visible part of the access package (see Figure 3-1). It also includes additional declarations that are needed by VMM but which would be tedious for the user to code explicitly. The package also defines procedures which build a symbolic description of the virtual records (a symbol table containing the character strings which name virtual records and their components) and which access symbol tables previously created and stored in the KAPSE database. These symbol tables support the reading and writing of the human-readable forms of VMM data structures. See Section 3.3.4 for a description of these forms. Any MAPSE tool using VMM must be compiled with the appropriate VMM access package.

3.2.4.2.2 Primitive Operations

The VMM access package contains a set of user-callable primitives that are used to create and access a VMM data structure. The operations declared in the access package are generally supported by more primitive operations defined in the VMM implementation package which are, in turn, supported by package DIRECT_IO in a layered fashion (see Figure 3-2).

These primitives are used to establish which data structure(s) will be referenced by the tool and to locate individual components within the structure to store, access, and manipulate actual data. The access package serves to collect together the operations for the applicable Virtual Record Types, and to provide an environment for default values.

3.2.4.2.3 Linear Representation

Linear representations of VMM data structures express VMM objects in terms of name-value associations. These linear forms can also be read by programs which use compatible virtual record types. This facility is used by the tool builder to display and generate instances of data structures during tool development and testing. Because the human readable form is simply a text file, it can be easily manipulated using a text editor.

The representation used for VMM data structures is loosely based on the form proposed for the external representation of the Diana intermediate language for Ada programs. This form is an adaptation of the linear graph notation used for the $TCOL_{Ada}$ intermediate language; the primary textual differences of interest are the use of explicit bracket tokens which permit nested forms, and a textual distinction between defining occurrences and references. The form used to represent VMM domains reflects the virtual memory subdomain structure explicitly and contains other constructs found to be useful in the linear graph notation used to represent virtual memory data structures in existing Intermetrics' compilers. This form of representation is called virtual record notation (VRN).

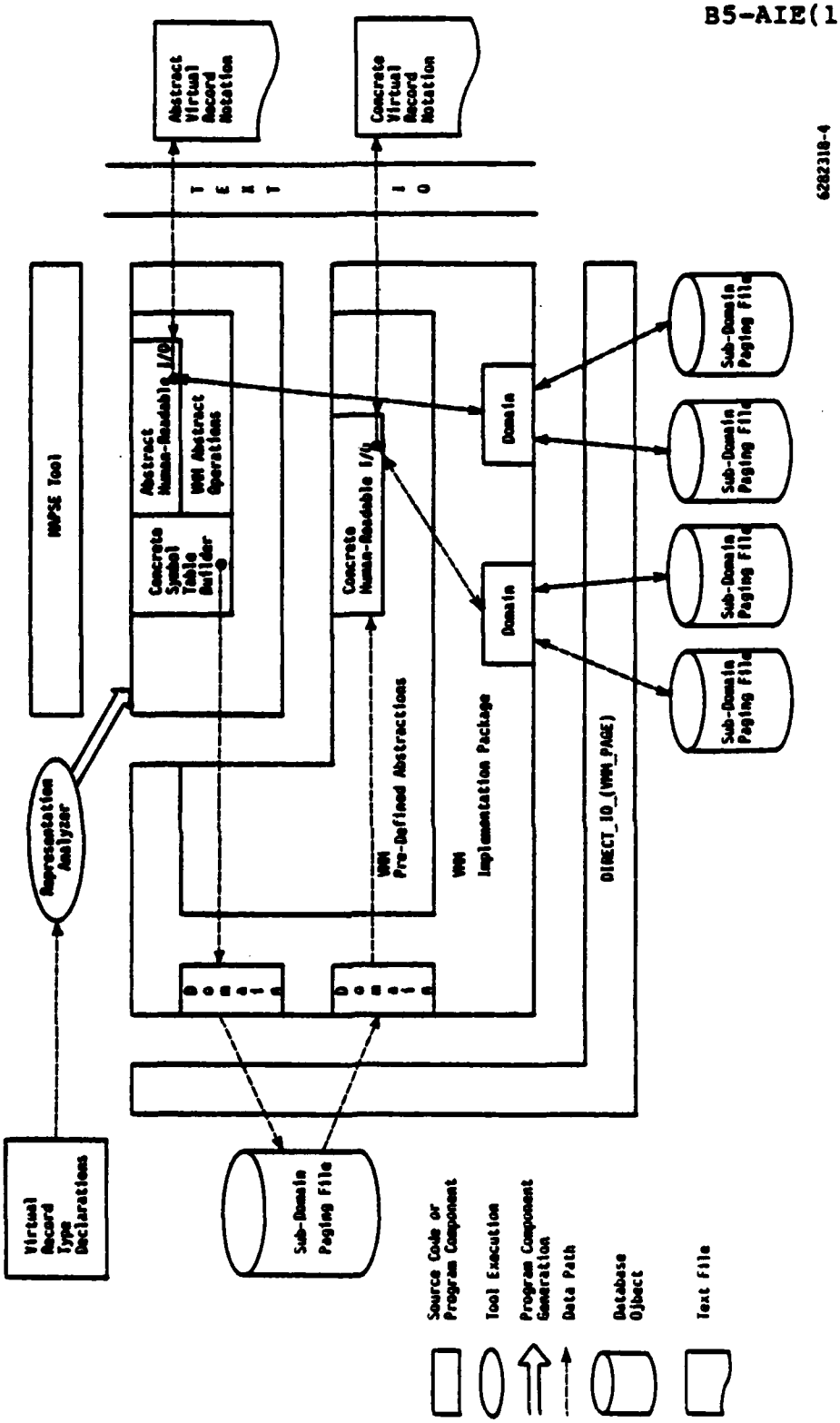


FIGURE 3-1: REP ANALYZER

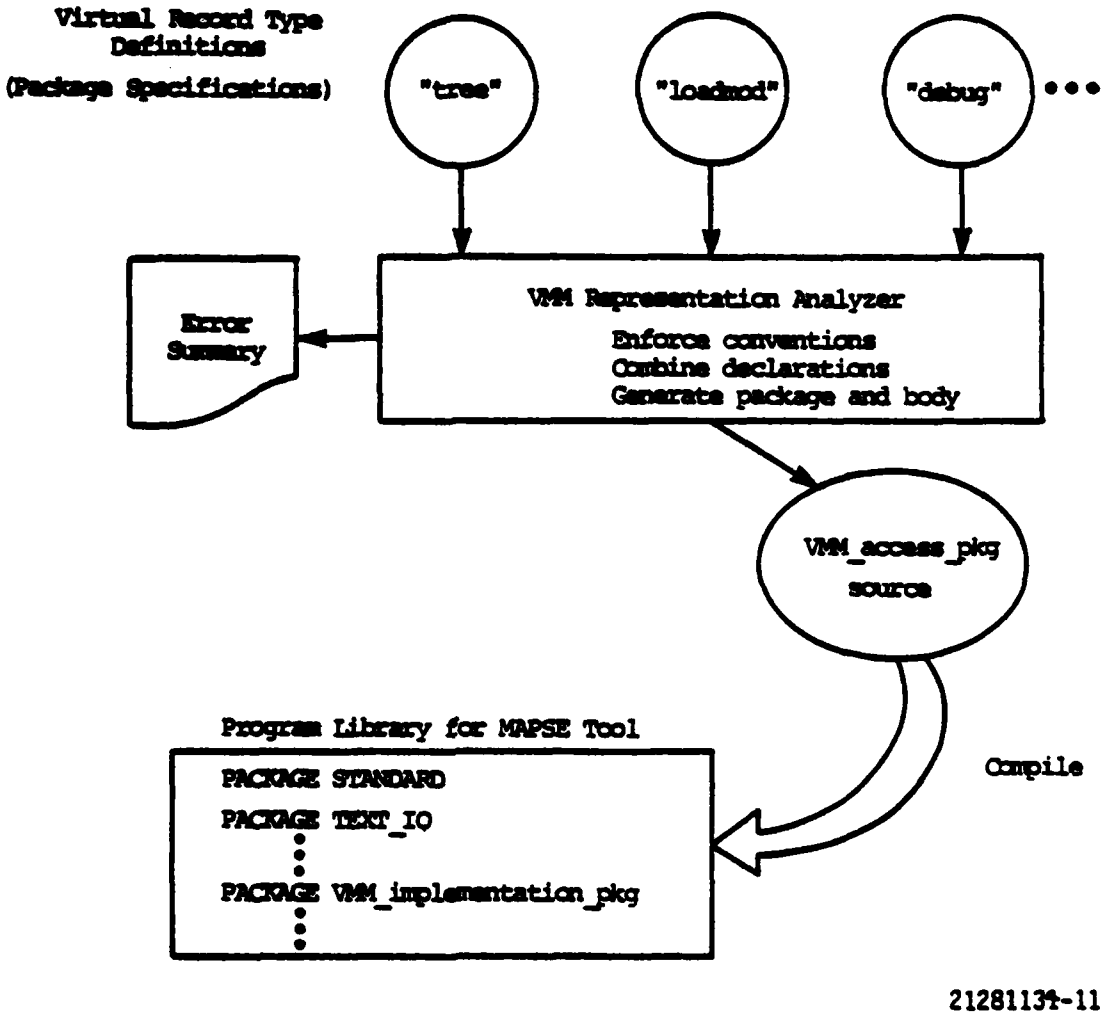


FIGURE 3-2: VMM PACKAGE HIERARCHY AND DATA PATHS

3.2.4.3 Ada Compiler Interface

The Ada ACCESS types which designate VMM objects can be specified as accessing checkpointed data by means of the MarkRelease compiler pragma. Such types are not subject to garbage collection or explicit storage reclamation that depends on the type of the object designated. Values for these types may thus be generated by unchecked conversions of an integer type suitable for address computations without interacting badly with Ada run-time storage methodology. For run-time realizations which do not use additional dope information to implement unchecked deallocation, the language-defined pragma "controlled" is used instead of "Mark Release".

The Rep Analyzer also has a direct interface with the compiler, in that it invokes the compiler front end (COMP.FE) through the compiler driven (COMP.FE.A) to perform syntactic and semantic analysis of the package specification it processes.

3.3 Detailed Functional Requirements

Detailed requirements for VMM are discussed below in terms of the four major components.

3.3.1 Rep Analyzer

3.3.1.1 Input

Input to the Rep Analyzer is a sequence of Ada package specifications defining the record types that define a VMM data structure. A single VMM data structure is a directed graph, constituting a <subdomain> (VMSD) that can, in several contexts, be accessed as a single entity. The nodes in the directed graph are VMM objects. The user defines nodes as <virtual records>, whose format is described later. The VMM system also includes several predefined data aggregates that are, in use, equivalent to user-defined virtual records. (For simplicity of discussion, the term virtual record will be used to denote VMM objects, with the distinction between user and system-defined made only where necessary.) Each VMSD contains VMM objects whose types are defined in one package specification input to the Rep Analyzer. Thus, each VMSD may be associated with a particular package name; that package name is the name of the Virtual Record Type of the VMSD.

VMSDs can include references (called VMM locators) to virtual records that reside in other structures VMSDs. To implement such inter-VMSD references VMM defines the concept of a domain - a collection of one or more VMSDs that may reference one another. Since VMSDs may have different Virtual Record Types, use of the VMM objects in a domain may require the use of declarations from different Virtual Record Types. Therefore, the collection of all

virtual record types that are to be known within a domain is specified to the Rep Analyzer, which then generates an access package containing the combined declarations for the specified Virtual Record Types. Even in the case of a domain that uses a single Virtual Record Type, an access package for that Virtual Record Type must be specified. An access package is specified in the Rep Analyzer input by a package specification containing a pragma which names the Virtual Record Types to be accessed.

A virtual record is realized in Ada as an object of some record type with a single discriminant of some enumeration type. The component types have statically determined sizes and contain no Ada ACCESS types or types for which assignment is not available. Thus each value of the discriminant names a virtual record "kind" with a statically determined size and representation; since it contains no ACCESS, TASK, or LIMITED types, objects of the type can be safely written to external files (i.e., KAPSE database objects accessed through package DIRECT_IO in PREDEFINED PACKAGES in KAPSE.RTS) and read by subsequent program activations. The central function of the VMM package involves the creation of VMM objects within external files and the support of direct references from one VMM object to another in an efficient manner whether the VMM objects reside in the same or different external files. (Note: the term external file is used in these sections to designate a KAPSE database object identified as a STRING to package DIRECT_IO, while the term object is used in the Ada sense; the term VMM object is used to designate VMM aggregates and virtual records which are allocated within external files by VMM operations and are identified by VMM locators).

An input Virtual Record Type package specification must include a WITH clause specifying VMM generic packages that include the declarations required to declare user-defined nodes and predefined data types. These generic packages will be defined in AIE(1).VMM(2).VMM. An input package specification which specifies the generation of an access package must name each Virtual Record Type package to be accessed in its WITH clause and also in the pragma "VRTYPES" which is processed by the Rep Analyzer (not by the compiler). In general, the input must be valid Ada, so any other unit dependencies must be identified by appropriate WITH clauses.

The subtypes for virtual record components are restricted to use a subset of the full type definition facilities of Ada. In particular:

1. No ACCESS, PRIVATE, limited or real types may be used.

2. Subtypes must be statically constrained. (All components must have statically determined sizes).

A type mark used in the declaration of a virtual record component can be considered to fall in one of the following categories:

1. A simple type. This category includes all discrete types.

2. A constructed type. This category includes records and arrays. It also includes statically constrained sub-types of the variable-length text string type.

3. A VMM locator type.

While the types in categories (1) and (2) are declared in the conventional manner in Ada, VMM locator types are declared by instantiating generic packages. One reason for this convention is that it allows the Rep Analyzer to associate additional information (the generic actual parameters) with virtual record components that contain VMM locators. This additional information includes such things as the type of elements in a VMM vector, list or set, or a default size for a VMM set. Such information is needed to support conversions to and from human readable forms. Another reason for the convention is that it allows the tool builder to associate a specific derived locator type for a supported abstraction with the type of element organized by the abstraction. The complete specification of these generic packages and how they are used is an implementation issue.

3.3.1.2 Processing

The Rep Analyzer reads one or more virtual record type definitions supplied as Ada package specifications, enforces the restrictions and conventions imposed by the VMM implementation, and generates Ada package specifications and package bodies that define an interface between the VMM implementation and tools using those virtual record types. (See Figures 3-1, 3-2).

Since the input to the Rep Analyzer is legal Ada, the bulk of the front end processing is done by invoking the Driver phase of the compiler COMP.FE.

The Diana tree is examined to verify that the conventions and restrictions for specifying virtual record types have been observed. If not, messages are added to any messages already attached to the compilation unit by the compiler. If no errors are detected by the compiler phases, and no deviations from the VMM conventions are found, new package specifications and package bodies are generated.

The package specifications generated by the Rep Analyzer are straightforward mechanical transformations of the input package specifications. For each package that defines a virtual record type, the following processing takes place:

1. Verify restrictions and conventions.

2. "Instantiate" generics using text substitution, producing non-generic packages in the output.

3. Add (overloaded) declarations for virtual record creation, locator dereferencing, attribute selection, and symbol table generation.

4. Add PRAGMAs and representation specifications where appropriate.

5. Generate a package body containing bodies for each of the subprograms declared in 3, above.

These functions require imparting to the generated package certain characteristics of virtual record types, such as the size of an object of each subtype and the offset of each component applicable to each subtype. This is done by generating expressions which use the predefined language attributes SIZE and POSITION, divorcing the Rep Analyzer from knowledge of the representation decisions that will be made by the compiler when the access package is compiled.

For each access package requested, a package specification and package body are generated as follows:

1. Generate an enumeration type with enumeration literals consisting of the name of each virtual record type package.

2. Generate each operation from each virtual record type definition package, declaring it in a package nested within the new package.

3.3.1.3 Output

The output from the Rep Analyzer consists of an error/summary listing and package specifications and bodies in source form. The error/summary listing contains any messages generated by the compiler phases used to process the input, as well as messages indicating any violations of the restrictions and conventions imposed by VMM. If there are no errors, a summary of the virtual record types may be optionally produced, showing for each virtual record kind the names and types of components applicable to it.

The package specifications output are named the same as input, and mention the appropriate VMM implementation packages in WITH clauses. The package bodies mention UNCHECKED_CONVERSION (in PREDEFINED PACKAGES in KAPSE.RTS) in WITH clauses and supply the bodies for those entities in the specifications which require them.

3.3.2 Abstract Data Types

VMM objects are typed objects in the same sense that Ada objects are typed. In general, a VMM object is designated by a typed VMM locator, where the type of the locator identifies the object as either:

a) a virtual record object with components defined by the user of the VMM implementation by means of virtual record type declarations; or

b) an object representing a particular data abstraction directly supported by the VMM implementation.

All such locator types are derived from a single locator type defined by the VMM implementation package, for which the locate procedure is defined.

The only operations available to the user of VMM are those defined in the visible part of the VMM access package generated by the Rep Analyzer. This access package contains dereferencing operations only for the locator type that designates user-defined virtual records. For the other locator types, only the appropriate operations for each abstraction are provided. Thus, while VMM locators can be seen as inherently "typeless" at the lowest level of implementation, indicating a position within an external file to be translated to a memory address within a buffer, derived types are used to establish compile-time verification of consistent locator usage. Further run-time checks are applied by both the compiled code (e.g. discriminant verification) and by the VMM implementation (e.g. virtual record type verification for each VMSD). In a fully-debugged tool, certain of these checks may be disabled to enhance performance.

3.3.2.1 Virtual Record Type

At the lowest level, a program examines and modifies a virtual record by dereferencing its locator and then operating on the components of the Ada record object designated by the ACCESS value so obtained. However, the VMM access package contains a function-procedure pair (attribute selectors) for each component of a virtual record, which is used to obtain/assign the value of a component. These are overloaded on the appropriate locator, pointer, and accessor types, allowing the choice of reference value and locking decisions to be made for performance tuning after the tool works correctly. (The relationship between locators, pointers, and accessors is explained in 3.3.3.)

In general, virtual record components are implemented as record components, and the attribute selectors will then be inline and have no additional runtime cost. It will also be possible to specify a complete implementation for virtual record components by writing procedure bodies for the attribute selectors. Besides simple scalar

values or references to other virtual records, virtual record components may contain statically-constrained arrays or text strings, nested records, or references to objects of an abstract data type supported by VMM.

The Rep Analyzer generates dereferencing and allocation functions for each virtual record type. These are specified in the visible part of the access package. An additional procedure is generated for each virtual record type to support the reading and writing of the human readable (VRN) representation of VMM data. This procedure takes a string parameter that names a database object; and it builds a symbolic description of the virtual record type within that database object. The symbolic description includes the (character string) name of each virtual record component applicable to each virtual record subtype, and a description of the type of each component sufficient to allow correct conversion between the internal value held within the component and a textual (VRN) representation of that value. The symbolic description is itself represented by VMM objects of a virtual record type defined by the VMM implementation. The database object containing the description of virtual record type is associated with a VMM VMSD that belongs to a domain which is managed by the VMM implementation independently from user-specified domains.

A number of operations that are applicable to virtual records use the same specification and body for locators designating objects of different virtual record types. The specifications and bodies for such operations are generally part of the VMM implementation package. The operations are derived by the derived type declarations in the access packages. Among these are operations to:

a) set root locator value for a VMSD;

b) obtain root locator value for a VMSD;

c) Iterate over all virtual records within a VMSD.

d) Output a sequence of domains, a sequence of VMSDs, or a single virtual record in virtual record notation.

e) Read virtual record notation into one or more domains.

3.3.2.2 Built-in Data Abstractions

VMM directly supports abstract (or encapsulated) data types for doubly-linked lists (chains), threaded lists of user-defined nodes, vectors (one-dimensional dynamic-sized arrays), uncounted sets, and variable-length strings. Direct support means that operations on the data types are defined by the VMM implementation, and that there is a distinctive representation defined for the type in virtual record notation. The fact that these types are directly supported

does not mean that other commonly used structures such as trees or dags (directed acyclic graph) cannot be defined as abstract data types; it simply means that the implementation of those types must be provided by the user of VMM in terms of the virtual record types on which they will operate, and that the VRN representation for objects of those types will show the structure by means of label references in virtual record components.

The directly supported types are not inherently more efficient than corresponding user-defined records. The primary reason for supporting these particular abstractions is the advantage gained in having a simple "standard" human-readable representation for them. Lists, vectors, and sets have a natural representation as a sequence of elements which, with the possible exception of vectors, is much more comprehensible than a lower-level exposition of the directed reference structures used to implement them. A quoted string is more pleasant to look at than a sequence of separate characters, besides being able to represent clearly a zero-length string. While trees can be expressed in a somewhat more comprehensible form (by nesting, indenting, or a more graphic means) the gain in clarity over a simple referential form is not nearly as great and quickly becomes lost when structures approach the size and depth likely to be encountered in actual tools, or when the form is adapted to include the more frequently encountered dag. Of course, supporting data abstractions which have a simple human-readable representation is not useful if the abstractions themselves are not useful. Extensive experience with compilers, linkers, separate compilation databases and other software development and support tools has shown the utility and effectiveness of the VMM-supported abstractions.

The following paragraphs briefly describe the functionality of each abstraction and the operations applicable to each. In general, there is a group of operations whose bodies depend upon the type of elements organized by the abstraction and another group whose bodies are fixed. The first group is generated by the Rep Analyzer for each element type defined to be organized by that abstraction in a virtual record type definition using attributes of the generic actual parameters which specify the element type. These operations generally invoke operations with fixed specifications which are parameterized by attributes of the elements such as size or position of information within the elements. Operations in the second group simply invoke fixed operations with type conversions applied to parameters and function return values ('Reference Manual for the Ada Programming Language' [6.4.1]). Operations in the second group may be thought of as operations "derived" from the types used to implement the abstraction. These types are defined in a fixed package which implements the data abstractions (the implementation package). Detailed specifications are given in AIE(1).VMM(2).VMM(1).

3.3.2.2.1 Vectors

The vector abstraction supported by VMM is similar to the Ada capability for dynamically-sized single-dimension arrays. However, VMM vectors are not simply mapped into corresponding Ada array objects since that would limit the size of a VMM vector to the size of a page buffer.

Operations are generated by the Rep Analyzer to create a vector, to access an element, and to obtain or set a value.

Operations are derived from the implementation package to obtain the size of a vector or to delete it.

3.3.2.2.2 Lists

VMM lists provide support for ordered sequences of values. There are two distinct models: doubly-linked circular chains ("predecessor lists") with with separately-allocated cells which hold element values, and singly-linked null-terminated threaded lists of user-defined modes. While predecessor lists naturally have operations that the threaded lists do not (e.g., remove an arbitrary element, change the value of an element, find the predecessor to an arbitrary element, or insert between an arbitrary element and its predecessor), the operations they have in common are specified such that they are interchangeable when a node appears no more than once in a list.

Operations are generated by the Rep Analyzer to create a list, create a cell, and to access a cell to obtain or set a value.

Operations are derived from the implementation package to locate the next or previous cell, to insert a cell, to remove a cell, to append or prepend one list to another and to delete a list.

3.3.2.2.3 Sets

VMM sets are modeled on uncounted sets with a realization that depends on the membership testing algorithm specified when the set is declared. The primary concern with regard to sets is the speed of finding a member; insertions, deletions, and even iterations over all members are not necessarily simple operations.

The algorithm for membership testing is specified by the generic package name input to the Rep Analyzer and is known as the equality type for the set. There are three basic equality types:

1. Bit vector equality. Applicable only to sets with elements of a discrete type; the realization is a bit vector, as in Pascal sets.

2. Value equality. Set members of a discrete type, a locator type, or a statically - constrained record, array, or string type. Set members are equal if their values are equal.

3. Association equality. Set members are considered to have two components: a key component and an associated value. The type of the key component is as for Value equality, above, and the comparisons are performed in the same way. The associated value must be of a locator type or a discrete type.

Operations generated by the Rep Analyzer include: creating new sets and adding members, finding the value of a member, finding specified values, and copying.

Operations derived from the implementation package include finding the next member in a set, the size of a set, the basic set operations (intersection, union, difference, symmetric difference), and set deletion.

3.3.2.2.4 Text Strings

VMM variable-length text objects are implemented much as in the TEXT_HANDLER package used as an example in the 'Reference Manual for the Ada Programming Language' [7.6]. Manipulations are performed using Ada objects of type character, type string, and type Text, with conversions from VMM text string objects (designated by locators) to text objects and assignments from text objects to VMM text string objects made explicitly when required. This makes effective use of overloading and the power of Ada functions for handling dynamically-sized objects, and also tends to reduce the total number of locator dereferences. Besides the allocate and delete operations there is only a single selection-function and update-procedure pair defined to operate on VMM text string objects. No specifications need to be generated by the Rep Analyzer. Operations are derived from the implementation package to create and delete strings and to access text values.

3.3.3 Basic Operations

A virtual record type defined via the Rep Analyzer is, in effect, a relocatable template. A given tool must invoke a variety of VMM primitives to store, retrieve and manipulate actual data within this template. A basic set of these primitives is devoted to locating a given node - that is, establishing a domain, the VMSD of interest, and, finally, the node of interest. Throughout this process, the aim is to find the node's location in virtual memory space (via VMM locators), to dereference the locator and obtain an access value that can be used to directly access an object in memory.

3.3.3.1 Creating Domains and Subdomains

A domain is a collection of one or more separate data structures (VMSDs). While defined as part of the same domain, independent structures may access one another.

In order to access VMM objects, a program must first create a domain via one of the primitive functions, obtaining a domain that is initially empty (has no VMSDs). VMSDs can be built in this domain by calling the appropriate function with parameters specifying:

1. Domain
2. Name of external file
3. Name of virtual record type
4. Mode of operation (read, update, extend, create)
5. List of segment numbers (optional)

This function only establishes the segment numbers (if provided) as ones that may be used to build the VMSD within the domain, and provides the information needed to access an external file when the VMSD is referenced; it returns a value which designates the VMSD. If the VMSD is never referenced, no database access occurs. If a segment number in the list is already available to the domain, an exception is raised. If the list is empty, segment numbers are assigned as necessary by VMM. While a domain may contain virtual records of various virtual record types, a given VMSD may only contain virtual records of the same virtual record type.

A virtual memory domain is characterized by:

1. The set of virtual record types that are known within it. These types are defined by the tool builder and processed by the Rep Analyzer to produce a single Ada package (a VMM access package) which includes the tool builder's type definitions. This characteristic is static.
2. The set of VMSDs of which the domain is composed. This characteristic is dynamic, in the sense that VMSDs are added to and removed from a domain by the execution of procedures, subject to the constraint that the virtual record type of each VMSD is known in the domain. The set of VMSDs which compose a domain defines a virtual address space in terms of VMM locators: each locator identifies a VMSD and a position within that VMSD.

3.3.3.2 Sub-domain Characteristics

A VMSD is basically characterized by an external file which is the repository for its data. Each such file has permanently associated with it a virtual record type and a list of segment numbers. The segment numbers of all VMSDs present in the same domain must be disjoint, i.e., given a segment number and a domain, at most one VMSD within that domain will be identified. Thus, from the point of view of a domain, a VMSD is identified by a domain and a segment number list. A segment number is either present or absent with respect to a domain. Each segment number present in a domain identifies a VMSD which, in turn identifies an external file, a virtual record type, and a mode of operation; it provides access to a virtual address space in terms of positions within the external file. The external file is accessed using a single instantiation of the generic package DIRECT_IO (in PREDEFINED PACKAGES in KAPSE.RTS) for elements of a record type defined by the VMM implementation as a VMM page: when the mode of operation for the VMSD is read-only, the file is opened with mode IN_FILE, and otherwise with mode INOUT_FILE.

Each VMM page is associated with an index value ('Reference Manual for the Ada Programming Language' [14.2]) that designates the element in the file which is assigned to store the page when it is not memory resident; this value is known as the page index. Read and write operations move the contents of a VMM page between the external file and a page buffer. Page buffers are assigned to the external files on a demand basis from a common pool of buffers allocated from the heap and shared by all VMSDs present in a program. Demand for a page buffer occurs when a page index within an external file is referenced, and that page index is not resident in a buffer currently assigned to the file. When demand occurs and all buffers in the common pool are already assigned, a buffer is selected for reassignment. If the contents of the page resident in that buffer have been modified since the buffer was assigned, the page is first written to its external file at the position indicated by the page index, and then it is reassigned to the page of the external file that was referenced.

When a VMM object is created within a VMSD, sufficient space for the object is located within the pages already assigned to its external file, or else a new page is created with a page index one greater than the largest page index assigned to the file. In either case, the space is reserved within a page and is designated by a VMM locator which is constructed from the appropriate segment number from the list for the VMSD which identifies the external file, the page index, and a position within the page. The position within the page is expressed as a VMM-defined integer type. The position is coded as an offset (in storage units) relative to the beginning of the page. The page index is coded as page number relative to the page index of the first page in the segment. Thus, three components are coded within the locator (segment number, page number, and offset) in order to represent the three logical components (VMSD, page index, and offset) without allocating a fixed addressing range

for each VMSD. Note that no simple VMM object (a user-defined node or any single data element organized by one of the abstract data types) can be larger than a page.

When a VMSD is added to a domain, its external file, virtual record type, and mode may be identified explicitly or a reference to an existing VMSD may be provided instead, creating an alternate list of segment numbers to access the same external file. The use of alternate segments is restricted to contexts in which locators which use the alternate numbers are dereferenced: no VMM objects can be allocated using the alternate segment numbers. The purpose of using an alternate number is to allow the automatic translation of VMM locators which reference a given VMSD into the corresponding VMM locators for a different VMSD (which is a new version of the first VMSD).

The automatic translation is specified by creating a translation VMSD which contains a VMM association set of locators. The set (called the translation set) has membership testing based on locators for objects in the first VMSD and associated values which are locators for objects in the new version. This translation VMSD is created by comparing the two versions, and since all three are then present in a domain at the same time, they all have distinct segment lists. Once the translation VMSD has been created, the original VMSD is removed and the translation VMSD is identified as such. The effect is that the segment list for the original VMSD (obtained from the translation set) is assigned to the translation VMSD, but dereference operations on VMM locators containing such segment numbers go through the translation set to obtain a new locator (with a different segment number) which is then dereferenced.

The primary motivation for translation VMSDs is to support the MAP utility described in AIE(1).PIF(1). This allows the recompilation of a unit without recompiling dependent units under certain circumstances. When the change to the recompiled unit is such that it would have no effect on dependent units already in the library, it may be desirable to be able to use dependent units without recompiling them (e.g., when only comments are changed). The MAP utility first verifies that the old and new units are sufficiently similar such that recompilation of dependents can be avoided. It then constructs a mapping from each VMM object in the old unit to the corresponding object in the new unit. This mapping is built as a translation VMSD which translates locators for objects in the old unit to locators for objects in the new unit. When this translation VMSD is operating in a domain, both the references to the old unit (occuring in dependents that were not recompiled) and references to the new unit will be correctly mapped by VMM to the appropriate objects in the new unit.

Each VMSD has a distinguished VMM locator, known as its root locator, which may be explicitly set or examined by VMM operations. Since all operations on VMM objects require a VMM locator to identify the object, the root locator generally identifies some object in the VMSD which provides linkage to all the other objects in that VMSD. However, since objects in one VMSD may be referenced from another, this is not necessarily so: one VMSD might serve as a directory for other VMSDs, in which case the directory VMSD might be the only one with a non-null root locator. Further, a VMSD which is temporary (i.e., its external file is created and deleted within a program activation) can be accessed by VMM locators held in program variables and need not provide for accessibility of its objects across program activations. In summary, a VMSD is characterized within a domain by (see Figures 3-3 and 3-4 for a logical representation of these relationships):

1. list of segment numbers;
2. external file of VMM pages;
3. virtual record type;
4. mode of operation;
5. set of pages defined for placement in the external file;
6. set of buffers assigned to hold resident pages;
7. root locator.

An external file that contains VMM data objects, but which is not seen in the context of an active domain and VMSD, is characterized by:

1. a list of segment numbers;
2. a virtual record type;
3. a root locator;
4. a set of pages;

3.3.3.3 Locator Model

When a locator is dereferenced, the first step is always to examine the characteristics of the segment it references and perform any indicated translation. This translation process has already been described in Section 3.3.3.2. This section, then is concerned with locators that have already had any needed translation applied.

VMM operations which involve VMM objects must have a method for locating each such object within external files and establishing the particular "kind" of VMM object so located. The method is that of defining the VMM locator type as a segment number and a position within that segment. (As described in Section 3.3.3.2, the segment number identifies both a VMSD and a position within that VMSD, while the page number and offset are relative to that starting position.) A locator has meaning only with respect to a dynamically created virtual memory domain: the VMM implementation maintains, during the course of a program activation, an association between a virtual memory domain and a number of segments, each of which identifies a VMSD within that domain.

A VMM locator is the only means of consistently designating a VMM object. A locator is, in many ways, similar to an Ada ACCESS value: it is a "typed pointer" with values generated by allocation operations, and a distinguished null value which designates no object at all. Locators within virtual records are used to implement data structures that can be conceptualized as attributed directed graphs in the same way that Ada ACCESS values within Ada record objects would be used. Functional differences between VMM locators and Ada ACCESS values are:

1. A VMM locator value generated by an allocation during a program activation can be written to an external file and then be read by a subsequent program activation and still be guaranteed to designate the same VMM object. By contrast, an Ada ACCESS value is only defined within the context of the program activation which performed the allocation.

2. The addressing range of a VMM locator is defined by the implementation of the VMM package, and is not dependent on host machine characteristics or on the size of the run-time heap available to the program activation.

Considering only the first distinction, the functionality of VMM locators could be achieved by defining conversion operations between external files of VMM objects with directed edges represented by VMM locators, and internal collections of Ada objects with directed edges represented by ACCESS values. This kind of approach has been used in some test-bed compiler implementations (e.g., TCOL, PQCC) where the external file representation is a sequential text file and each compiler phase reads, and converts to internal form, the output from the previous phase, operates on the internal form, and then writes a text file representation to be read by the next phase. Another approach to achieving the first functionality avoids using ACCESS values at all; instead, graph nodes are represented by elements of a single array of variant records, with directed edges stored as array indices (e.g., AIDA implementation). Although this offers the advantage that no conversion processing is needed when reading or writing an external file, memory space utilization becomes a critical issue in the

design of the record variants since the array elements will be allocated the space required for the largest variant: optimal utilization occurs when each variant is the same size. In addition, using array indices to implement pointers results in a loss of clarity and run-time efficiency, motivating the design of ACCESS types in the Ada language in the first place.

The second functional aspect of VMM locators is not addressed by the original implementation strategies for either TCOL or AIDA : both rely on sufficient memory resources to contain the internal representations for all nodes that are used during a graph traversal.

While the direct addressability of both the VM/370 and OS/32 systems is sufficient to accommodate intermediate representations for useful Ada programs, an implementation of the compiler and separate compilation facility that assumed adequate memory to access all nodes as Ada objects could not be rehosted to a system with a smaller address space without seriously impacting processing capacity. The processing capacity on the OS/32 system would also be significantly less than on VM/370.

Further, experience with a separate compilation facility for a Pascal dialect has shown that even the 24-bit addressing range of VM/370 imposes compile-time limits that can be exceeded by compilations of large applications programs. Therefore, the model for VMM locators requires, as an integral part of their use, a discipline which provides nearly complete independence from memory and addressing constraints imposed by the host machine. Not only does this requirement establish the portability of MAPSE tools to smaller machines, but it removes the absolute limits on processing capacity inherent in any program which relies on memory addressing to access an internal database. The actual addressing range for locators is an implementation choice; one possibility considered a candidate is to implement locators to address 32767 segments, each with up to 32 pages, and each page containing 2048 bytes. Such a locator value could be represented in 32 bits and would permit a wide range of VMSD sizes without wasting address bits.

3.3.3.4 Mapping Locators to Objects

The VMM strategy uses locators which have the same representation internally within an executing program as they do on an external file - in that sense similar to the AIDA approach using array indices. However, instead of implementing each element of the data structure as an element of an Ada array and using an array index to provide access, the VMM approach implements a VMM object as a block of storage units within a page buffer. The ADDRESS pre-defined attribute is used to convert an ACCESS value for the buffer to an implementation-defined integer type to which an offset is added before being converted to an ACCESS value of a type appropriate to the VMM object being accessed (by means of UNCHECKED_CONVERSION in PREDEFINED PACKAGES in KAPSE.RTS. The advantages of this approach are basically two-fold:

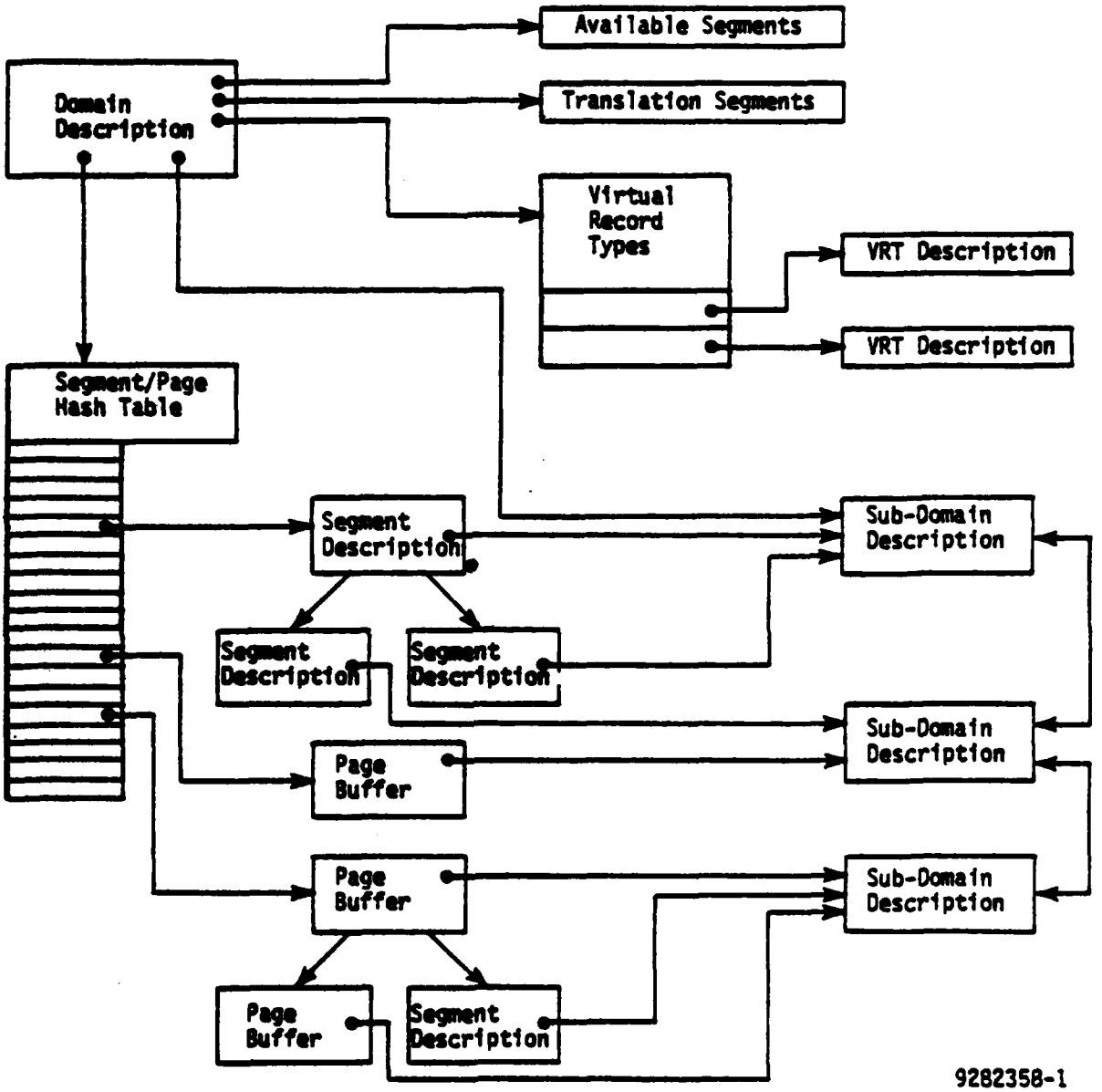


FIGURE 3-3: DOMAIN STRUCTURE

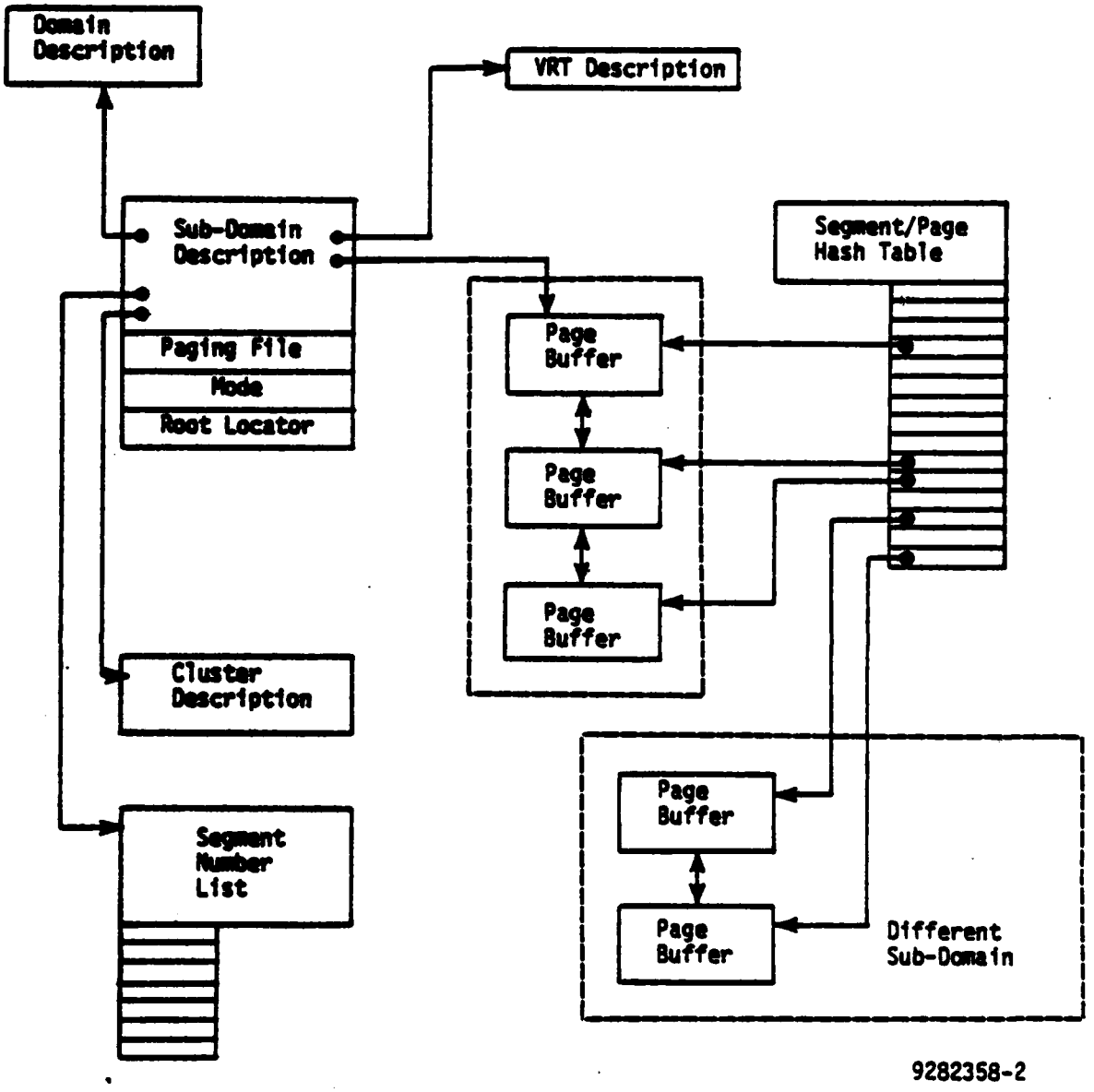


FIGURE 3-4: VMSD STRUCTURE

1. The relative sizes of the record variants to be accessed do not affect the efficiency of storage utilization: the number of VMM objects accessible within a buffer is determined by the minimum space required for each object and not by a worst-case fit based on the largest variant.

2. Once an ACCESS value is computed for a VMM object, references to its components do not require repeated indexing operations.

Recalling that a VMM locator encodes a segment number, a page number (relative to the segment) and an offset (relative to the page) the mapping from locator value to ACCESS value is accomplished in two steps:

1. Locate a page buffer assigned to the segment which contains the specified page, returning an ACCESS value for the buffer.

2. Compute the ACCESS value for the object as ObjectType(Buffer.all'address+offset) where ObjectType is an instantiation of the generic function UNCHECKED_CONVERSION with input type an implementation-defined integer type and result type an ACCESS type for a VMM object.

In terms of the VMM implementation, a single "locate" procedure takes IN parameters consisting of the domain and the locator value and produces OUT parameters consisting of the ACCESS value for the buffer and the value Buffer.all'address+Position. The locate procedure is defined in the visible part of the VMM implementation package, and is thus visible to the VMM access package produced by the Rep Analyzer (since the analyzer places its name in the WITH clause for the access package). Since the tool using the VMM access package does not normally have the implementation package in its visibility list, it only makes use of the locate procedure by means of the dereferencing functions defined in the access package. These functions return values of an unconstrained ACCESS type for a variant record type. (In general, the tool will not use this facility directly, but use the selector functions and procedures of Section 3.3.2.1 to fetch and store the values of virtual record components.) Thus, the instantiation and use of UNCHECKED_CONVERSION is limited to the package (body) generated by the Rep Analyzer and is not used or seen by the tool which uses VMM. Furthermore, the VMM implementation package verifies the virtual record type for each VMSD (in terms of the character-string representation of the virtual record type name once, when the external file is first accessed.

The dereferencing functions generated by the Rep Analyzer for each virtual record type then use the ACCESS value for the page buffer (returned by "locate") to verify that the "integer" being converted to an ACCESS value is in fact the address of a VMM object that was created in a VMSD specified to contain objects of that same virtual record type. Thus, the use of UNCHECKED_CONVERSION by the

VMM access package to achieve efficient storage utilization does not compromise Ada type safety. Subsequent component selection using the "converted" ACCESS value is again subject to the normal discriminant verification applied at run-time to record variants designated by unconstrained ACCESS values.

3.3.3.5 Dereferencing

The process of obtaining an Ada ACCESS value in order to perform operations on a VMM object is called "locator dereferencing". Since locators define an address space which is not constrained by the addressing and memory-size limits of the run-time model for Ada objects, it is clear that the memory available for Ada objects will, in general, only be able to represent a subset of the VMM objects accessible to a program through locators. The subset of VMM objects resident in memory at any given time must include at least those objects that are named by ACCESS values in Ada statements. However, a dereference operation which identifies a VMM page that is not resident in memory at that time, may need to reuse space previously assigned to a VMM object, causing the ACCESS value computed for that object to become invalid: dereferencing drives the demand paging mechanism. Since the algorithm that determines which resident objects (i.e., which page buffer) will be replaced during a dereference operation cannot predict the future pattern of use for previously computed ACCESS values, it is always possible that the replaced objects would include those which were about to be referenced again. In the absence of further interaction with the dereferencing and underlying paging mechanism, only the most recently computed ACCESS value could be considered valid at a given point in a program execution; the ACCESS values invalidated by that computation (if any) cannot be determined by the tool builder. Figure 3-5 shows the basic steps involved in locator dereferencing.

Every use of data held within a VMM object could require a locator dereference to obtain an ACCESS value guaranteed to be the most recently computed value: dereferencing can be viewed as being analogous to a "load" instruction on a single accumulator machine (actually, rather than "load" and "store", VMM defines "load read-only" and "load for update" operations with an implicit "store" carried out when a value loaded by the second form is about to be overwritten). Using the analogy, it should be clear that the performance of a program using VMM depends critically on the speed of the dereference operation, and that providing a capability analogous to a multiple-register architecture would allow significant performance enhancements to be made. VMM defines a method for reducing the cost of dereference operations in certain cases by splitting the operation into two separable parts called pointer computation and pointer dereference; a capability called dereference locking allows a program to force a VMM object to remain resident and accessible by the same ACCESS value throughout a specified region of program execution, effectively allowing full access to more than one VMM object at a time (these locked references are called VMM accessors).

Note that these operations are essentially optimization techniques that can be employed by the tool builder to improve the performance of a program. Operations on typed locators are generally overloaded on corresponding pointer and accessor types such that code written using locators can be modified to use pointers or accessors with minimal textual change (e.g., enclose the code in a block which renames the locator object and then declares a pointer or accessor object with the identifier of the locator).

3.3.3.6 Pointer Computation

The mapping from a VMM locator to an ACCESS value was described as a two-step process: (1) locate a page buffer containing the required page from the specified VMSD and (2) perform the address arithmetic. Of the two steps, the first is much more costly. Even when the desired page is already resident in a buffer and no I/O operations are required, finding that buffer from the locator value alone implies some type of indexing or searching operation. Given the locator realization considered earlier, it is clear that direct indexing using the segment number and page number would not be practical, requiring some type of search. While the use of simple hashing techniques can make the search time quite short, just computing a hash may take longer than the time needed to access the data once the buffer is found. A pointer computation converts a VMM locator (in the context of a VMM domain) to a form that already has had translations applied and which "remembers" intermediate results from the last time it was dereferenced. In particular, a VMM pointer contains:

1. VMM domain value;
2. VMM locator value;
3. ACCESS value for a page buffer;
4. An address within the page buffer (implementation-defined integer);
5. A sequence number.

The first four values are obtained by performing any necessary translation on the input locator value and then invoking the "locate" procedure to compute the third and fourth values. The timestamp value is simply copied from the page buffer.

Each page buffer has a sequence number component which is updated by the page buffer methodology routines whenever the buffer is reassigned to a different page or deassigned from active use (e.g., when a VMSD is removed). The update consists of incrementing a single counter which is maintained for the entire pool of page buffers, and then copying the incremented value to the particular page buffer being reassigned. The range of the counter is such that it could never overflow during the course of a single

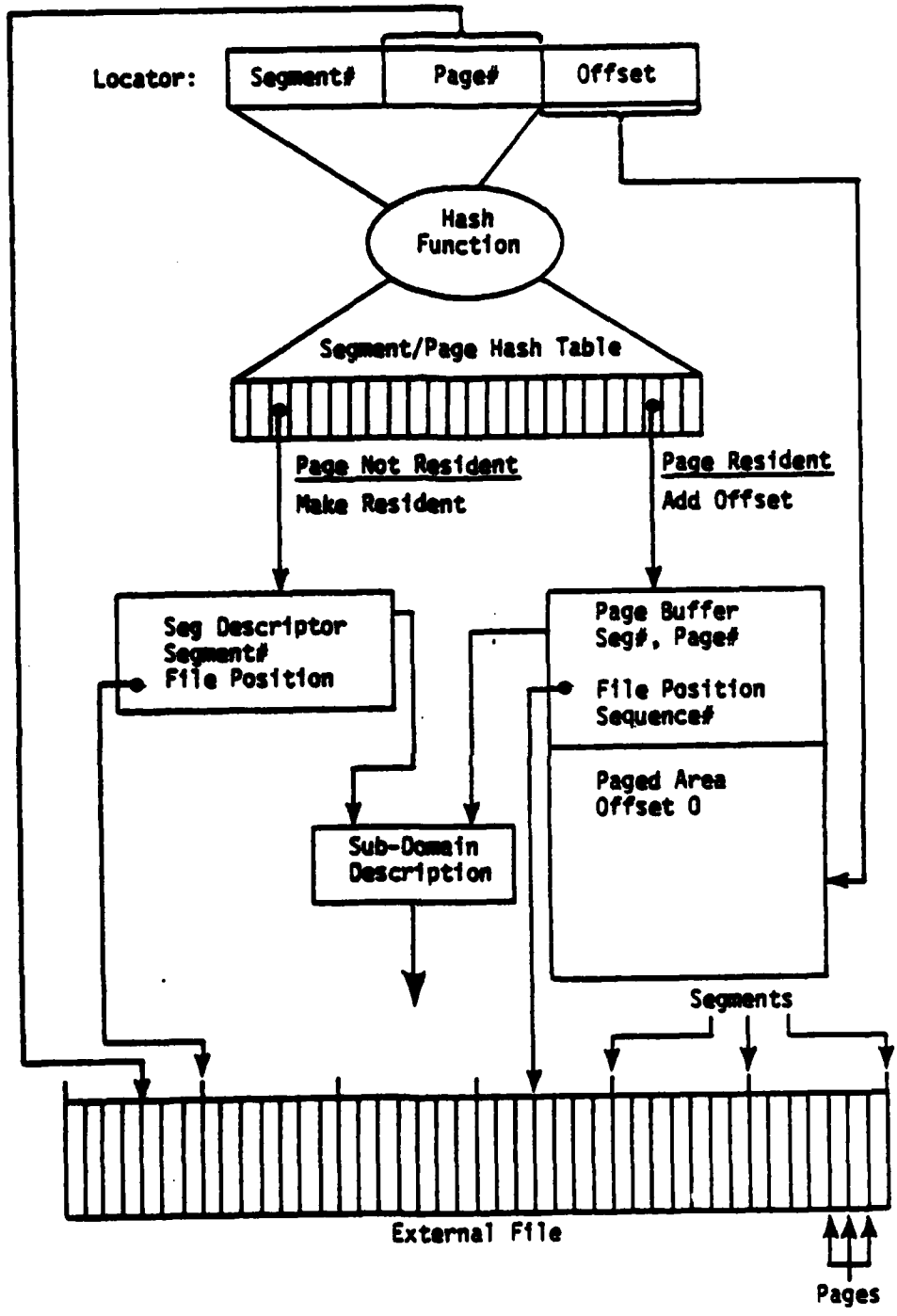


FIGURE 3-5: LOCATOR DEREFERENCING

program activation; it is initialized once (by elaboration of the library unit for the VMM implementation package) and never reset. (A 32-bit integer is adequate for this purpose). The result is that the sequence number recorded in the buffer designated by a VMM pointer will equal the sequence number recorded in the pointer itself if and only if the same page has remained resident in that buffer since the pointer was computed.

3.3.3.7 Pointer Dereferencing

Once a VMM pointer has been computed from a VMM locator, dereferencing the pointer to obtain an ACCESS value can be accomplished with no hash computation or searching at all as long as the VMM object remains resident. If the object has not remained continuously resident since the pointer was computed, the cost of the dereference is no greater than the cost of dereferencing the original locator, and may be less (i.e., if the original locator required translation). Furthermore, the processing is so simple that pointer dereference functions can be specified as inline (via the language-defined PRAGMA), with the result that no procedure invocation is required when the object is resident.

3.3.3.8 Dereference Locking

When a dereference is locked, the VMM object that is designated by the dereference is forced (by the buffer methodology routines) to remain resident in the same buffer, and thus remain accessible by the same ACCESS value for as long as the lock remains in effect. This capability allows a program to safely designate more than one VMM object by an ACCESS value; at any point in a program, the VMM objects that can be designated by ACCESS values are those that are locked plus the most recently dereferenced value.

A VMM procedure, overloaded on locators and pointers, establishes a locked dereference, producing a VMM accessor which designates the same VMM object as the input locator or pointer. Operations on VMM accessors require no hashing and no validation since the designated object is guaranteed to remain resident until the accessor is explicitly unlocked.

3.3.4 VMM Virtual Record Notation I/O

This facility supports the input and output of data in a human-readable form called Virtual Record Notation (VRN). A MAPSE tool that operates on a virtual memory domain can convert the entire domain or a set of VMSDs to or from VRN, using operations declared in the VMM access package. (The various input subprograms are collectively known as the "reader", while output subprograms comprise the "writer"). VRN preserves the semantics of VMM data structures, including the distribution of virtual records among VMSDs and explicit sharing of data values. All values (not just

nodes) are optionally labelled. This provides a means to explicitly represent that objects are identical (i.e., not two different objects containing the same value). A domain containing an active translation VMSD (Section 3.3.3.2) would be output without any explicit representation of the translation VMSD. Any references to the "from" VMSD which it translates would be translated to references to the "to" VMSD. On input, the structure is re-created without a translation VMSD. Alternatively, if a translation VMSD is present but not active (i.e., it has not been specified to begin automatic translation for its alternate segment numbers), the translation VMSD is output normally; on input the complete structure is re-created and the translation VMSD could be activated.

The syntax of VRN is derived from the syntax proposed for an external representation of Diana with some significant departures. These are motivated by the fact that VRN describes a data structure implementation as well as a data abstraction. The concrete notation explicitly shows how the data structure is realized in terms of Ada records, database objects, and the set of data abstractions implemented by VMM. Thus, each VMM object is associated with the database object in which it is stored. On the other hand, the precise mapping between VRN and VMM objects is based on name associations determined by the Rep Analyzer. A grammar for VRN is given in 10.1.

Referring to the grammar, it can be seen that VRN is defined as a sequence of virtual memory domains, and that a virtual memory domain is represented as a sequence of VMSDs. Each domain first declares a list of VMSD specifications that it will use in reference to virtual records, a FREE_SPEC, an ASSIGNED_SPEC, an ABSOLUTE_SPEC, and a USE_SPEC. Each VMSD declaration associates a label for a VMSD with components that identify a database object for that VMSD and a virtual record type description produced by the Rep Analyzer. An optional list of segments to use for the VMSD can be specified.

A FREE_SPEC is a list of segment numbers which can be used in the domain. An ASSIGNED_SPEC is a list of assigned segment numbers for the domain. An ABSOLUTE_SPEC specifies a list of absolute labels for the domain. An absolute label contains the segment number, page number and offset to be used to locate a data object. A USE_SPEC associates an ordinary string label with an absolute label.

The writer uses VMM basic operations to iterate over VMSDs and nodes (virtual records) within a domain. The symbolic description of virtual records, called the symbol table, is used to determine attributes (components) each node along with their type, size, and range. The symbol table describes virtual record types and specifies all of the node kinds for each type. For each node kind it creates a character string for the identifier and describes all of the possible attributes for that kind. Attribute values are output by converting data at the node locator's address to the specified data type.

The reader takes a file of VRN as input and either creates the domains and VMSDs represented in the input or adds to existing domains and VMSDs. The symbol table is used to determine the allowable attributes for each node along with their type, size and range. Constraint checks are made on the data values read in and then the values are written to a cell which is allocated for that node.

Following the declarations are representations of the virtual records within each VMSD (called VMSD definitions). The Root identifies the root object of that VMSD. Each DEF represents a VMM object that is located within that VMSD or it defines a cluster (a group of VMM objects allocated "together" with respect to the paging file).

A virtual record object (node) is represented by an identifier followed by components (attributes) enclosed in special brackets. The identifier names a discriminant value used in the virtual record description for the VMSD. That discriminant value identifies the names and types of all components that may be part of the virtual record. Each component associates a value with a name. The name must be one of those component names that are applicable to the discriminant value, and the value must be compatible with the type defined for that component in the virtual record description. While the discriminant value restricts the names and types of components that may appear within the record's delimiting brackets, it does not require that each possible component be explicitly represented. Those components that have internally the value they are assigned when a virtual record is created (i.e., the default value) need not appear in the external representation.

There are two forms of VRN-concrete and abstract, both adhering to the same syntax. Concrete VRN represents virtual records in the order in which they were created. Nodes are output sequentially within each VMSD and the node labels are a hex representation of the node locators. A VMSD label is the first segment number for that VMSD. In abstact VRN, VMSDs and nodes have decimal numbers as labels. Nodes are sorted within a VMSD according to node kind or cluster number.

3.4 Capacity

The overall capacity of VMM (i.e., the number of buffers, structures, domains) is constrained by the heap usage requirements of the program using VMM. The following capacities are the minimal performance standards.

(1) $(2^{31}-1)$ *8 bits of information per domain (17,179,869,176).

(2) 256 VMSDs accessible at one time within a single domain.

(3) $(2^{23}-1)$ *8 bits of information per VMSD (67,108,856).

(4) simple objects at least as large as 7680 bits

(5) 1 domain with controlled segments in addition to 4 domains without controlled segments.

(6) 4 clusters per VMSD

(7) 8 virtual record types per domain

(8) 256 distinct node kinds per virtual record type.

(9) Locator dereference is fewer than 20 machine instructions when the referenced object is memory resident and all optimizations have been applied (inline code, no constraint checking).

(9) Pointer dereference is fewer than 10 machine instructions when the referenced object is memory resident and all optimizations have been applied.

(10) Accessor dereference in fewer than 5 machine instructions when all optimizations have been applied (the object is guaranteed to be memory resident).

4.0 QUALITY ASSURANCE PROVISIONS

4.1 Introduction

A quality product is assured by a combination of sound design, careful implementation and effective testing. This section identifies specific issues of quality that are significant for VMM. The levels of testing to be conducted for VMM are as follows:

(1) Computer subprogram (CPC) testing for the Rep Analyzer (VMM.VMM.A), Abstract Data Types (VMM.VMM.B), Virtual Record Notation Input/Output (VMM.VMM.C), and Basic Operations (VMM.VMM.D).

(2) Computer program testing (CPCI) for the complete VMM.

(3) Subsystem acceptance testing for the complete VMM.

(4) Integration testing for VMM.

4.2 Test Requirements

Computer subprogram tests are tests of the Computer Program Components (CPC). They are designed to verify the specifications presented in the C-5 documentation [C5-AIE(1).VMM(1)]. These include integration tests to verify interfacing among the CPC's. All computer subprogram tests are designed and executed by the implementation personnel; test descriptions and test reports will be submitted to Quality Assurance (QA) after testing.

Formal Test Procedures define the functional computer program testing for VMM. These tests must verify that VMM meets the requirements presented in this document. Specifically, these tests must verify:

(1) that the Rep Analyzer generates correct and compilable virtual record type declaration and access packages;

(2) that the Rep Analyzer correctly detects and reports user errors;

(3) correct operation of paging mechanism;

(4) allocation and dereferencing operations;

(5) domain and subdomain operations;

(6) data abstraction definitions and operations;

(7) VRN input/output operations.

These tests will be executed when VMM is completed using the Bootstrap compiler, and after VMM is compiled using the production compiler. Formal tests for VMM will be defined in formal test procedures. These tests will be executed: once after the production compiler is complete and has been rehosted to the Kapse; a second time, when the full Kapse is complete; and, finally, when the Kapse is rehostd to OS/32. The primary formal test of VMM is to build the Ada compiler and execute the Ada Compiler Validation Capability (ACVC) tests. In addition, formal tests will contain a set of development tests provided by the implementors, a set of functional tests, and a test to verify the detection and reporting of errors using the critical record type Diana with errors introduced. Test reports for these formal tests will be issued by QA.

Integration testing for VMM must verify the interfaces between VMM and any subsystem or CPCI which it uses and the interface between VMM and the tool builder. Thus the integration tests will verify the interfaces between: the Rep Analyzer and the compiler phases LEXSYN and SEM in COMP.FE; VMM and direct and text external file facilities in KAPSE.RTS; and VMM and specific tool users. In addition, VMM will be used during compiler development, once the production compiler is available, during all development compilations. This use will provide continual informal integration testing for VMM.

Formal integration testing will consist of the following procedures:

Step 1. The Rep Analyzer will be invoked using the virtual record type definition (see 3.3.4) as input.

Step 2. The virtual record type declaration and access packages generated in Step 1 and the test program will be compiled where the test program first calls the procedure which builds a symbol table description for the virtual record type in Step 1 and then calls the Virtual Record Notation output procedures. (The specifications and bodies for the procedures in the test program are in the access package.)

Step 3. Finally the test program is executed and the Virtual Record Notation which was created by the test is examined.

In Step 1, the Rep Analyzer uses the compiler phases in COMP.FE and the external file facilities of KAPSE.RTS; In Step 2, the user tool interface is used, since for any compilation the compiler uses VMM objects and operations as does the program library tools (which are invoked by the compiler). In Step 3, the text file facilities of KAPSE.RTS are used by the VMM operations.

4.3 Acceptance Test Requirements

Since the VMM subsystem is designed primarily for use by the compiler and the program library, it alone does not address any of the initial contract requirements except by virtue of being available for use by future MAPSE tools. Therefore, the functional accceptance tests for VMM will be tests to verify the use of the Rep Analyzer and its error reporting and to verify the operations in the access packages which are generated. These tests will be a subset of the functional computer program tests described in Section 4.2.

The performance tests for VMM will be designed to verify the VMM capacity and performance data presented in this document. These tests will demonstrate that the capacities specified can be supported by creating objects, subdomains and domains with the specified sizes or properties and either accessing the objects or invoking the domain and subdomain operations (e.g., listing the subdomains in the domain). The performance of dereference operations will be evaluated by examining the generated code from the compiler.

5.0 NOTES

None.

APPENDIX A: VIRTUAL RECORD NOTATION GRAMMAR

The following grammar describes both the concrete and abstract forms of virtual record notatation. It is specified in a simple variant of BNF formalism which is acceptable to the YACC parser generator program on the PWB/UNIX Time Sharing System. Note that comments are embedded in the grammar using /* ... */ conventions. The VRN itself has a commenting convention that is not described in the grammar. The commenting convention differs from Ada, allowing comments to be embedded within lines, and to span lines: comments are both introduced and terminated by a vertical bar (or exclamation point, depending on the available character set). This convention allows programs which generate virtual record notation to place comments beside a token without altering the line structure, and it allows a human reader to "comment out" portions of lines easily when using VRN for debugging and testing purposes.

```
/*****************************************************************/
/* In grammar rules, reserved words and special characters       */
/* are in lower case within apostrophes, while token classes     */
/* recognized by the lexer appear in lower case and syntax       */
/* categories appear in upper case.  When a syntax category or a */
/* token class is prefixed by a lower-case descriptive word with */
/* underscore, the descriptive word provides a semantic hint     */
/* regarding the use of that item in the surrounding context,    */
/* but is not of consequence in parsing.                         */
/*****************************************************************/

/*****************************************************************/
/* The following name token classes recognized by the lexer.     */

 % token id integer quoted arrow lbrack rbrack

/* id and integer are as defined by Ada Lexical rules.           */
/* quoted is a doubly-quoted string of characters with IDL       */
/*              escape sequences for non-printing characters.    */
/* arrow is a "reference" indicator, ':' or '<-' depending on    */
/*            available printing characters.                     */
/* lbrack and rbrack are '[' and ']' or '(.'^'.)', as above.     */
/*****************************************************************/

%%
READABLE:
    'abstract' ASCII |
    'concrete' ASCII |
    ;
ASCII:
    'ascii' 'is'
            DOMAIN_DECL_PART
    'begin'
            DOMAIN_DEF_PART

    'end' 'ascii' ';' ;
DOMAIN_DECL_PART:
    DOMAIN_DECLS |
    ;
DOMAIN_DECLS:
    DOMAIN_DECL |
    DOMAIN_DECLS DOMAIN_DECL ;
DOMAIN_DECL:
    'domain' domain_id 'declares'
    FREE_SPEC ';' |
    ASSIGNED_SPEC ';' |
    ABSOLUTE ';' |
    USE_SPEC ';' |
          SUBD_DECL_PART
    'end' domain_id ';'
SUBD_DECL_PART:
    SUBD_DECLS |
    ;
SUBD_DECLS:
    SUBD_DECL |
    SUBD_DECLS SUBD_DECL ;
SUBD_DECL:
```

```
SUBD_DECL:
    subdomain_LABEL ':' 'subdomain' lbrack
                                'object' '=>' STRING ';'
                                /* identifies "paging file" */
                                'nodes' '=>' STRING ';'
                                /* identifies virtual record type */
                                'Segments'=>'('segment_LABLIST')' ';'
                                /* identifies available segments*/
                            rbrack ;
LABEL:
    id |
    integer ;
LABLIST:
    LABEL |
    LABLIST ',' LABEL ;
FREE_SPEC:
    'free' '(' segment_LABLIST ')' ;
ASSIGNED_SPEC:
    'assigned' '(' segment_LABLIST ')' ;
ABSOLUTE:
    'absolute' '(' node_LABLIST ')' ;
NODE_SUBD_LABEL:
    'node' node_LABEL |
    'subdomain' subdomain_LABEL ;
USE_SPEC:
    'for' NODE_SUBD_LABEL 'use' integer;
DOMAIN_DEF_PART:
    DOMAIN_DEFS |
    ;
DOMAIN_DEFS:
    DOMAIN_DEF |
    DOMAIN_DEFS DOMAIN_DEF ;
DOMAIN_DEF:
    'domain' domain_id 'is'
            SUBD_DEF_PART
    'end' domain_id ';' ;
SUBD_DEF_PART:
    SUBD_DEFS |
    ;
SUBD_DEFS:
    SUBD_DEF |
    SUBD_DEFS SUBD_DEF ;
SUBD_DEF:
    'subdomain' subdomain_LABEL 'is'
            ROOT ';'
            DEF_PART
    'end' subdomain_LABEL ';' ;
ROOT:
    REF |
    DEF ;
REF:
    'null' |
    arrow REF_LABEL ;
REF_LABEL:
    LABEL |
```

```
    ATTRIBUTES ;
DEF_PART:
    DEFS |
    ;
DEFS:
    DEF |
    DEFS DEF ;
DEF:
    node_LABEL ':' NODE ';' |
    'cluster' cluster_LABEL ';';
NODE:
    node_id |
    node_id ATTRIBUTES ;
ATTRIBUTES:
    lbrack ATTR_PART rbrack ;
ATTR_PART:
    ATTRS |
    ;
ATTRS:
    ATTR |
    ATTRS ATTR;
ATTR:
    attr_id '=>' VALUE ';' ;
VALUE:
    UNLABELED_VALUE |
    VALUE_LABEL ':' UNLABELED_VALUE ;
VALUE_LABEL:
    simple_LABEL  |
    INFORMATIVE_LABEL ;
INFORMATIVE_LABEL:
    '<*' LABEL_INFO simple_LABEL '*>' ;
LABEL_INFO:
   INFO_PARTS |
   ;
INFO_PARTS:
   INFO_PART |
   INFO_PARTS INFO_PART;
INFO_PART:
   'subdomain' '=>' subdomain_LABEL ';' |
   'cluster' '=>' cluster_LABEL ';' ;
UNLABELED_VALUE:
   'true' | 'false' |
   NODE |
   integer |
   STRING |
   REF |
   AGGREGATE |
   SEQUENCE ;
AGGREGATE:
   '(' SIZE_PART ELEMENT_PART ')' ;
SIZE_PART:
   lbrack size_LABEL rbrack |
   lbrack size_LABEL '..' size_LABEL rbrack|
   ;
ELEMENT_PART:
   ELEMENTS |
   ;
```

```
ELEMENTS:
    ELEMENT |
    ELEMENTS ELEMENT ;
ELEMENT:
    'others' '=>' VALUE ';' |
    VALUE '=>' VALUE ';';
SEQUENCE:
    '<' SIZE_PART, VALUE_PART '>' ;
VALUE_PART:
    VALUES |
    ;
VALUES:
    VALUE |
    VALUES VALUE ;
STRING:
    quoted |
    STRING '&' quoted ;
```

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840